

Constraints and Triggers

Instructor: David Martin

Reference:

*A First Course in Database Systems,
3rd edition, Chapter 7 (but not section 7.4)*

Constraints and Triggers

- A *constraint* is a relationship among data elements that the DBMS is required to enforce.
 - *Example*: key constraints.
- *Triggers* are only executed when a specified condition occurs, e.g., insertion of a tuple.
 - Easier to implement than complex constraints.

Kinds of Constraints

- Keys/Unique constraints
- Foreign-key, or referential-integrity constraints
- Value-based constraints
 - Constrain values of a particular attribute
- Tuple-based constraints
 - Relationship among components of tuple
- Assertions
 - Any SQL boolean expression (not implemented in most relational DBMS, not discussed much in this lecture)

Review: Single-Attribute Keys

- Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- Example:

```
CREATE TABLE Beers (  
    name    CHAR(20)  UNIQUE,  
    manf    CHAR(20)  
);
```

Review: Multi-Attribute Key

- The bar and beer together are the key for Sells:

```
CREATE TABLE Sells (  
    bar    CHAR(20) ,  
    beer   VARCHAR(20) ,  
    price  REAL,  
    PRIMARY KEY (bar, beer)  
);
```

Foreign Key

Sells

bar	beer	price
Above AVG	Heineken	6.00
Above AVG	Sapporo	6.00
Chugs and Slugs	Budweiser	4.00
Chugs and Slugs	Sapporo	4.00
Chugs and Slugs	Heineken	5.00
Hi Bar	Heineken	3.00
Hi Bar	Heineken	4.00
McGinty's Pub	Molson Golden	5.00
Select Clientele	Heineken	7.00

Beers

name	manf
Heineken	...
Sapporo	...
Budweiser	...
Heineken	...
Molson Golden	...

Foreign Keys

- Values appearing in attributes of one relation must also appear together in specific attributes of another relation.
- **Example:**
 - In `Sells(bar, beer, price)`, we might expect that a `beer` value also appears in `Beers.name` (the name column of the Beers table, the primary key for that table).

Declaring Foreign Keys

- Use keyword REFERENCES, either:
 1. After an attribute (for one-attribute keys)
 2. As an element of the schema:
FOREIGN KEY (<list of attributes>
REFERENCES <relation> (<attributes>)
- Referenced attributes must be declared as either PRIMARY KEY or UNIQUE.

Foreign Key Declared With an Attribute

```
CREATE TABLE Beers (  
    name CHAR(20) PRIMARY KEY,  
    manf CHAR(20) );
```

```
CREATE TABLE Sells (  
    bar CHAR(20),  
    beer CHAR(20) REFERENCES Beers(name),  
    price REAL );
```

Foreign Key

Declared as a Schema Element

```
CREATE TABLE Beers (  
    name CHAR(20) PRIMARY KEY,  
    manf CHAR(20) );
```

```
CREATE TABLE Sells (  
    bar CHAR(20),  
    beer CHAR(20),  
    price REAL,  
    FOREIGN KEY(beer) REFERENCES  
        Beers(name) );
```

Example: Adding Foreign Key

```
CREATE TABLE Beers (  
    name CHAR(20) PRIMARY KEY,  
    manf CHAR(20) );  
CREATE TABLE Sells (  
    bar CHAR(20),  
    beer CHAR(20),  
    price REAL );
```

```
ALTER TABLE Sells  
ADD FOREIGN KEY (beer)  
REFERENCES Beers(name);
```

Foreign Key

Sells

bar	beer	price
Above AVG	Heineken	6.00
Above AVG	Sapporo	6.00
Chugs and Slugs	Budweiser	4.00
Chugs and Slugs	Sapporo	4.00
Chugs and Slugs	Heineken	5.00
Hi Bar	Heineken	3.00
Hi Bar	Heineken	4.00
McGinty's Pub	Molson Golden	5.00
Select Clientele	Heineken	7.00

Beers

name	manf
Heineken	...
Sapporo	...
Budweiser	...
Sagres	...
Molson Golden	...

Enforcing Foreign-Key Constraints (Referential Integrity, RI)

- If there is a foreign-key constraint from *referring relation R* to *referenced relation S*, then violations may occur two ways:
 1. An insert or update to *R* introduces values not found in *S*, or
 2. A deletion or update to *S* causes some tuples of *R* to “dangle”

Actions Taken --- (1)

- **Example:** suppose $R = \text{Sells}$, $S = \text{Beers}$.
 - That is, Sells refers to Beers
- An insert or update to Sells that introduces a nonexistent beer must be rejected.
- A deletion or update to Beers that removes a beer value found in some tuples of Sells can be handled in one of three ways (next slide).

Actions Taken --- (2)

Available Actions When S is Modified

1. *Default*: Reject the modification.
2. *CASCADE*: Make the same changes in Sells.
 - Deleted beer: delete Sells tuple.
 - Updated beer: also change value in Sells ...
 - ... so that Sells.beer has the same new value as Beers.name
3. *SET NULL*: Change Sells.beer to NULL.

Example: CASCADE

- Upon Delete of the Bud tuple from Beers:
 - Delete all tuples from Sells that have beer = 'Bud'
- Upon Update of the Bud tuple by changing 'Bud' to 'Budweiser':
 - Change all Sells tuples that have beer = 'Bud' to have beer = 'Budweiser'

Example: SET NULL

- Upon Delete of the Bud tuple from Beers:
 - Change all tuples of Sells that have beer = 'Bud' to have beer = NULL.
- Upon Update of the Bud tuple, changing 'Bud' to 'Budweiser':
 - Make the same change to tuples of Sells that have beer='Bud' as for deletion (making beer=NULL).

Choosing a Referential Integrity Policy

- When we declare a foreign key, we may choose policies SET NULL or CASCADE independently for deletions and updates.
- Follow the foreign-key declaration with:
`ON [UPDATE, DELETE][SET NULL, CASCADE]`
- Two such clauses may be used, one for UPDATE and one for DELETE
- Otherwise, the DEFAULT (Reject) is used.

Example: Setting Policy

```
CREATE TABLE Sells (  
    bar CHAR(20),  
    beer CHAR(20),  
    price REAL,  
    FOREIGN KEY (beer)  
        REFERENCES Beers(name)  
        ON DELETE SET NULL  
        ON UPDATE CASCADE  
);
```

Attribute-Based Checks

- Constraints on the value of a particular attribute.
- Add CHECK (<condition>) to the declaration for the attribute.
- The condition may use the name of the attribute, but references to any other relation or attribute name must be in a subquery.

Example: Attribute-Based Check

```
CREATE TABLE Sells (  
    bar CHAR(20),  
    beer CHAR(20) CHECK ( beer IN  
        (SELECT name FROM Beers) ),  
    price REAL CHECK ( price <= 5.00 )  
);
```

Example: Named Constraints

```
CREATE TABLE Sells (  
    bar CHAR(20),  
    beer CHAR(20) CHECK ( beer IN  
        (SELECT name FROM Beers)),  
    price REAL  
        CONSTRAINT price_is_cheap  
        CHECK ( price <= 5.00 )  
);
```

```
ALTER TABLE Sells DROP CONSTRAINT price_is_cheap;  
ALTER TABLE Sells ADD CONSTRAINT price_is_cheap  
    CHECK ( price <= 5.00 );
```

Timing of Attribute-Based Checks

- Attribute-based checks are performed only when a value for that attribute is inserted or updated.
 - **Example:** `CHECK (price <= 5.00)` checks every new price and rejects the modification (for that tuple) if the price in Sells is more than \$5.
 - **Example:** `CHECK (beer IN (SELECT name FROM Beers))` is not checked if a beer is deleted from Beers (unlike foreign-keys).

FOREIGN KEY vs. Attribute CHECK

- `CREATE TABLE Studio (....
 presC# INT REFERENCES MovieExec(cert#)
);`
- `presC# INT CHECK
 (presC# IN (SELECT cert# FROM MovieExec))`
- With Attribute CHECK
 - No Referential Integrity policies
 - Deletion / update on MovieExec is ignored
 - Reject NULL value for presC# when cert# has no NULL

NOT NULL

- Special case of attribute-based check

```
CREATE TABLE Sells (  
    bar    CHAR(20),  
    beer   VARCHAR(20),  
    price  REAL NOT NULL,  
    PRIMARY KEY (bar, beer)  
);
```

- Consequences
 - Cannot insert a tuple unless a value is specified or there's a non-null default
 - Cannot use the SET NULL policy to fix foreign-key violations

Tuple-Based Checks

- CHECK (<condition>) may be added as a relation-schema element.
- The condition may refer to any attribute of the relation (same tuple).
 - But references to any other tuples or relations require a subquery.
- Checked on insert or update only.

Example: Tuple-Based Check

- Only Joe's Bar can sell beer for more than \$5:

```
CREATE TABLE Sells (  
  bar      CHAR(20),  
  beer     CHAR(20),  
  price    REAL,  
  CHECK (bar = 'Joe's Bar' OR  
         price <= 5.00)  
);
```

Assertions

- These are database-schema elements, like relations or views.
- Defined by:
 CREATE ASSERTION <name>
 CHECK (<condition>);
- Condition may refer to any relation or attribute in the database schema.
- (Not implemented in most Relational DBMS because it's too expensive.)

Constraints: Summary

- Keys/Unique constraints
- Foreign-key, or referential-integrity constraints
- Value-based constraints
 - Constrain values of a particular attribute
- Tuple-based constraints
 - Relationship among components of tuple
- Assertions
 - Any SQL boolean expression (not implemented in most relational DBMS, not discussed in this lecture)

Triggers: Motivation

- Assertions are powerful, but the DBMS often can't manage them efficiently
 - ... and they're frequently not implemented by DBMS's.
- Attribute- and tuple-based checks are checked at known times, but are not that powerful.
- Triggers let the user (often the DBA) decide when to check for any condition.

Event-Condition-Action Rules

- Another name for “trigger” is an *ECA rule*, or *event-condition-action* rule
- *Event* : typically a type of database modification, e.g., “insert on Sells”
- *Condition* : Any SQL boolean-valued expression
- *Action* : Any SQL statements

Preliminary **Example**: A Trigger

- Instead of using a foreign-key constraint and rejecting insertions into **Sells(bar, beer, price)** with unknown beers, a trigger can add that beer to Beers, with a NULL manufacturer.

Example: Trigger Definition

```
CREATE TRIGGER BeerTrig
```

```
  AFTER INSERT ON Sells
```

The event

```
  REFERENCING NEW ROW AS NewTuple  
  FOR EACH ROW
```

```
  WHEN (NewTuple.beer NOT IN  
        (SELECT name FROM Beers))
```

The condition

```
  INSERT INTO Beers(name)  
    VALUES(NewTuple.beer);
```

The action

CREATE TRIGGER

- Either:

CREATE TRIGGER <name>

- Or:

CREATE OR REPLACE TRIGGER <name>

- Useful if there is a trigger with that name and you want to modify the trigger.

Options: The Event

- AFTER INSERT can also be BEFORE INSERT.
 - Also, can be INSTEAD OF, if the relation is a view.
 - A clever way to execute view modifications is to have triggers translate them to appropriate modifications on the base tables.
- INSERT can also be DELETE or UPDATE.
 - And UPDATE can be UPDATE ON a particular attribute.

Options: FOR EACH ROW

- Triggers are either “row-level” or “statement-level.”
- FOR EACH ROW indicates row-level; its absence indicates statement-level.
- *Row level triggers* : Execute once for each modified tuple.
- *Statement-level triggers* : Execute once for a SQL statement, regardless of how many tuples are modified.

Options: REFERENCING

- INSERT statements involve a *new* tuple (for row-level) or *new* “table” (for statement-level).
 - The “table” is the set of inserted tuples.
- DELETE involves an *old* tuple or table.
- UPDATE involves both *old* and *new*.
- Refer to these by
[NEW OLD][ROW TABLE] AS <name>

Options: The Condition

- Any boolean-valued condition.
- Evaluated on the database state that exists before or after the triggering event, depending on whether BEFORE or AFTER is used in the event clause.
- Access the new/old tuple/table through the names in the REFERENCING clause.

Options: The Action

- Affects the database state that exists before or after the triggering event, depending on whether BEFORE or AFTER is used in the event clause.
- There can be more than one SQL statement in the action.
 - Surround by BEGIN . . . END if there are more than one.
- But queries make no sense in an action, so we are really limited to modifications.

Another Example

- Using `Sells(bar, beer, price)` and a unary relation `RipoffBars(bar)`, maintain a list of RipoffBars that raise the price of any beer by more than \$1.

The Trigger

